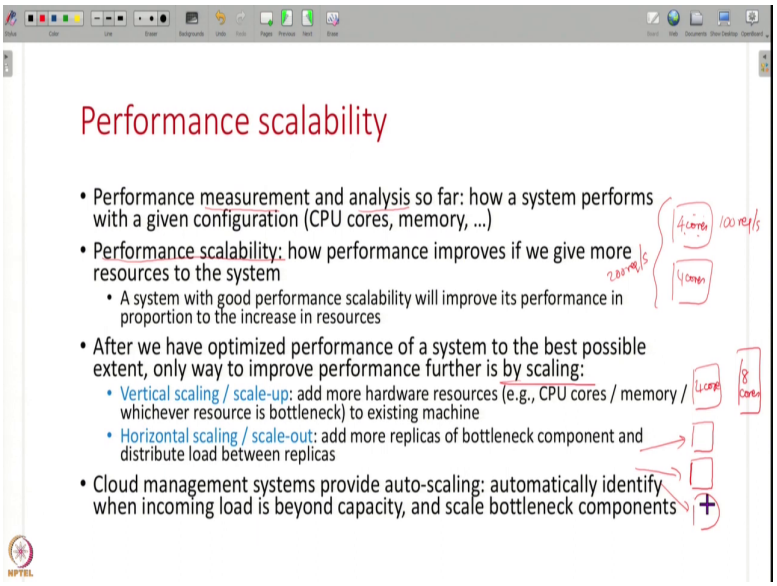
**Design and Engineering of Computer Systems**
**Professor Mythili Vutukuru**
**Computer Science and Engineering**
**Indian Institute of Technology, Bombay**
**Lecture - 49**
**Performance Scalability**

Hello, everyone, welcome to the thirty fifth lecture in the course design and engineering of computer systems. In this lecture, we are going to wrap up our discussion on performance engineering by covering the last topic, which is performance scalability. So let us get started.

(Refer Slide Time: 0:35)




So, what is performance scalability? Let us understand that first. So, far what we have seen this week is how to measure performance, how to analyze the performance that we have got and how to kind of optimize the performance of a system. A system with a given number of CPU cores and memory, how to optimize its performance?

So, the other aspect of performance is what is called performance scalability. So, the performance of a system only tells us what performance you get for a given set of resources, this system has some capacity of say 100 requests per second. Now, over 4 cores, the system has 100 requests per second. Now, performance scalability tells me that if I add 4 more cores into the system, will the total capacity become 200 requests per second that is what I expect.

I have given more CPU cores and all the CPU was fully utilized. So, I gave it 4 more cores now is my capacity also increasing. So, can a systems capacity increase if we give it more resources? So, if it does, if performance improves, if we give more resources, then the system

is set to have good performance scalability. On the other hand, if no matter how many resources you give the performance does not improve then the system has poor scalability.

Not that a system can have very high performance itself to begin with, but the performance can scale poorly. So, these are two different things performance measurement itself, and the performance scalability these are two different aspects. Now, once we have this system with given configuration given hardware, we have optimized its performance using all the techniques studied in the previous lectures and we can optimize it no more, but we still need more performance, this performance is not enough it is not meeting the offered load, the expected incoming load.

In such cases, what we do is the only way to improve performance is by scaling by adding more resources and hoping that the performance improves because of these extra hardware resources. And the scaling can be done in two ways, which is one is what is called vertical scaling, you have a 4 core CPU, I will add more CPU cores, I will make it 8 core CPU, or I will add more memory or whatever is the bottleneck resource, you will add more of that this is called vertical scaling or scale up, you are simply making one machine more and more powerful.

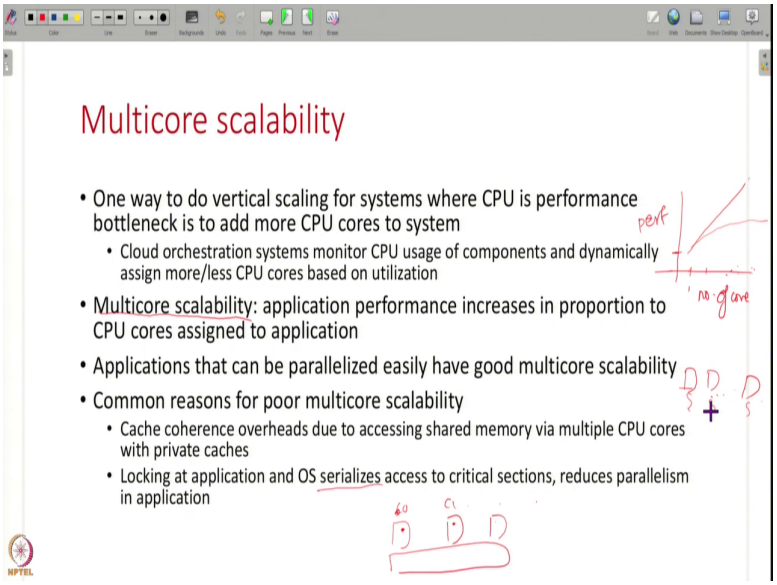
The other is what is called horizontal scaling or scale out. So, if this component is the bottleneck, I will add run the software on another machine or another VM and create a replica of this component and split my traffic amongst these replicas. If this component can only handle 100 requests per second, but I am getting 200 requests per second, I will have a second copy of it to split the look, this is called horizontal scaling or scale out.

So, both of these are tried in general in order to scale the performance of a system. Now, note that a lot of cloud management orchestration systems, this auto scaling they do it automatically, if some component has become the bottleneck, then they will identify, the CPU's at 100 percent utilization, then they will, add an extra replica and so on, all these cloud orchestration software's they will measure the incoming load and they will measure your utilization and automatically scale up components when needed.

So, this auto scaling, dynamically expanding your resources when your load increases and also similarly scaling down if you do not need so many replicas shutting some of them down. All of this is very important in a real-life system, because you do not know how much load is

going to come at any point of time. Therefore, you have to dynamically adapt scale your system to meet the incoming load.

(Refer Slide Time: 4:26)




So, first, let us understand a little bit more on vertical scaling. Specifically, suppose your system the performance is limited by the CPU your CPU cores are 100 percent utilized, and you want to add more CPU cores in your system to improve performance. When you do this, what you need is you need a property called multi core scalability, that is the performance of your application should increase in proportion to CPU cores. Otherwise, vertical scaling is meaningless it is not beneficial to add more CPU cores if the performance stays the same even if I give extra CPU cores.

So, what can you do to have good multi core scalability this is important to know? Note that this kind of scaling is done all the time, once again cloud orchestration systems, they will monitor your CPU usage and, give more CPU cores to your VM or take away some CPU cores. So, this dynamically scaling the number of CPU cores given to an application component is very commonly done.

But just giving more CPU cores is not enough, if your application's performance does not increase with increasing core support for one CPU core, you are getting this much performance with two and so on, then your application's performance should linearly increase. This is the performance with the number of cores, this should roughly increase like this only then is it beneficial to add more cores, otherwise, if it does not increase, and there is no point adding more cores.

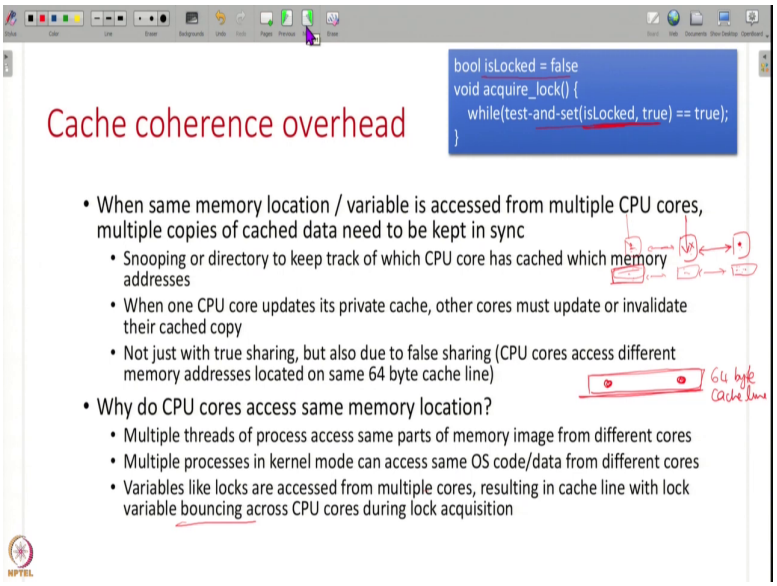
So, applications that can be parallelized easily, the application logic can easily if it is running on one core, it can easily be replicated on the other course such applications have good multi core scalability. But why do some applications not have good multi core scalability? For some applications no matter how many cores you add, the performance will more or less plateau out at some level, they cannot use more CPU cores beyond the point. Why does this happen? There are many reasons.

One is of course, cache coherence overheads. Each CPU core has its own private caches, and it has some common caches. So, we have seen this before. So, if you have multiple CPU cores, all of them accessing the same memory, then the CPU core has accessed some memory location, then the CPU core also wants it then there is some cache coherence traffic.

So, this cache coherence overheads means that you cannot just keep on adding CPU cores forever to your system, at some point, you want to run out of more CPU cores is not going to improve performance. So, cache coherence is one important reason while the performance of an application may not scale beyond a certain number of course.

The other reason is also within the application there could be a lot of serialization, just because you have so many parallel cores does not mean the application threads can run in parallel, maybe this thread is waiting for this thread for some reason for some locking, things like that there could be some serialization and not enough parallelism in an application. So, these are the common reasons for poor multi core scalability.

(Refer Slide Time: 7:27)



So, let us understand this in a bit more detail. So, when do these cache coherence overheads occur? They occur when the same memory location is accessed from multiple CPU cores, so this CPU core has a private cache, this CPU core has a private cache, and this CPU core is accessing say, some memory location x here and the CPU core also wants to write to the same memory location.

Then you need some synchronization here to either update the latest value here or invalidate this value fetch it from here, something needs to be done, you need some snooping some directory to keep track of which core has which memory location and when one CPU core updates its copy in its private cache, all other cores, which also have a copy of this data have to either update their copy or invalidate their copy.

So, these invalidation messages need to go to all the other cores, which also have used this same memory location in the past. So, these all these invalidation messages, updating messages, the snooping, directory, all of this imposes overhead, this is all extra work. Therefore, you cannot just keep on adding CPU cores to your system and expect that the performance will improve.

And this is not only a problem with true sharing, this is also false sharing. 2 CPU cores could be accessing different memory locations that are located on the same 64-byte cache line. Recall that, caching is done at the granularity of not bytes, but chunks of 64 bytes each. Therefore, different cores are using different data, but they are all on the same cache line. So, the same cache line has to be invalidated here and invalidated here when it is updated here.

So, this causes cache coherence overhead. So, then the question comes up why do CPU cores need to access the same memory location? Why cannot this thread access different memory this thread access this different memory. So, sometimes multiple threads may be accessing the same parts of a memory image, they might be accessing the same global variable in of the process or the same variable in the heap of the process.

So, then in such cases, the same memory location may be requested by different threads that are running on different CPU cores, which belong to the same process. The other thing that can happen is multiple processes on kernel mode. So, the kernel mode and data structures is the same. When any process goes into kernel mode, it is accessing the same PCBs and same list of processes and everything in such case is also the same OS code might be requested here and here and multiple different CPU cores by different processes in kernel mode.

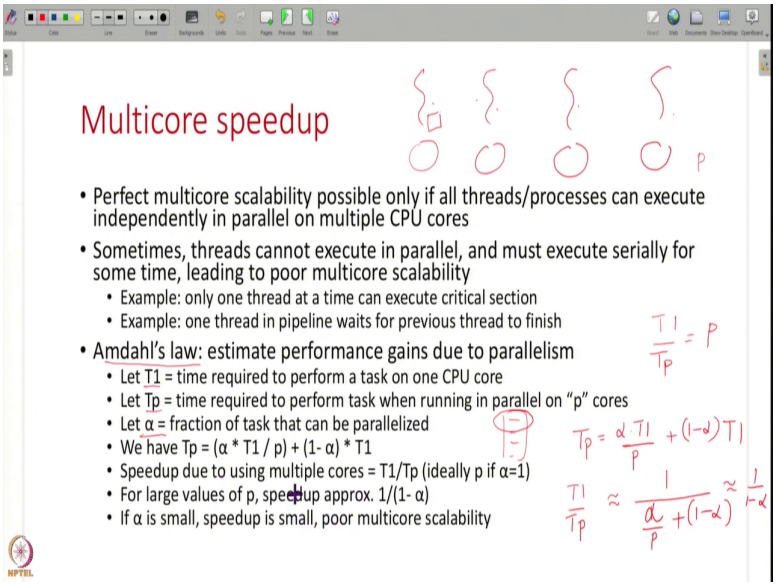
The other thing is, commonly threads need things like locks. And this locks are also accessed from multiple CPU cores. For example, this is the code we have seen off how to acquire a lock, you have this lock variable, we have seen this way back in this course, and different threads when they try to acquire the lock, what they do is they try to set this lock variable to true. And if the previous values also true, then you have not acquired the lock you wait, so this is the code to acquire a spin lock using this hardware instruction that basically updates this lock variable.

Now, this is locked, this variable the cache line containing this variable, suppose this thread has written to it, then this thread also running on another core also wants the lock it is also trying to do test and set right to this lock. Another thread on another core is also trying to access the lock trying to acquire the lock, all of these threads are contending for the lock. And what is happening? This cache line that has this variable, it is being updated here, then again, some other core will update this copies invalidated, some other core will update again, this copy is invalidated.

So, this cache line keeps this is called bouncing, the same cache line with multiple threads on different cores are writing to the same variable or trying to access the same variables than this cache line keeps bouncing across these different cores keeps getting updated here invalidated here, updated here, invalidated here, this keeps on happening and this causes a lot of extra traffic and overhead in the system.

So, locks because the lock variable is shared across different CPU cores, this will happen. In general, this cache coherence overhead hurt the performance of your system, especially when you want to scale to a large number of CPU cores.

(Refer Slide Time: 11:54)




And the other problem is that sometimes threads and processes cannot execute in parallel all the time. Just because you have n threads and n CPU cores, it may not be the case that all of these threads can always run in parallel on each CPU core, it is not possible. Why? because maybe this thread has entered a critical section and it has acquired a lock, then all these other threads cannot run in parallel.

Why? Because they are waiting for the lock or maybe it is a pipeline design where this thread has to do something only then this thread can do something else, they are waiting on a condition variable for whatever reason, it is not possible that all the code in your program can be executed in parallel on multiple CPU cores. Therefore, it is foolish to expect that as you add more CPU cores your performance will just keep on linearly increasing in proportion to the number of CPU cores.

In fact, there is a law or a simple formula and not just a complicated law, but a simple formula which is called the Amdahl's law that tells you how much performance gains you can get due to parallelism. Now, suppose I have some application that is doing some work, and some work takes the time T1 on one CPU core and if the same work I try to do it in parallel by having multiple threads on p different CPU cores, it is taking a time of Tp.

So, ideally, what do I expect? I expect that this T1 / Tp this is called my speed up, if I have 1 core, I took 100 seconds, if I have 10 cores, I took only 10 seconds each then I got a perfect scale up. This should ideally be equal to p, because now the same work you are doing it

across multiple different cores and if it is parallelizable, you will get a speed up equal to p. But in real life, this is not the case, why?

Because your entire application code cannot be parallelized. Suppose in your application logic only some part of it can be parallelized, the other part cannot be parallelized, it is a critical section it has to be run by the threads one after the other. So, if α is the fraction that can be parallelized, then what do you have when you run on p cores, this α part of your work it can be parallelized.

So, instead of this α part instead of taking α * T1 time, it is taking α * T1 / p time because you are running it in parallel on these p cores. Then the remaining 1 - α part what is happening? You are not able to parallelize it, it will still take the same amount of time as it took for a single core. Therefore, this is your formula for Tp and the speed up T1 / Tp if you do this, it will come out to not exactly p.

And if you do this T1 / Tp and if you take for large values of p, what is it will work out to? This 1 / (α / p + 1 - α) that is what your speed up will work out if you divide T1 by T p. And now for large values of p what is happening? This is approximately equal to 1 by 1 minus alpha. So, what is this saying? If you have large parts of your code that can be parallelized, your alpha is close to 1.

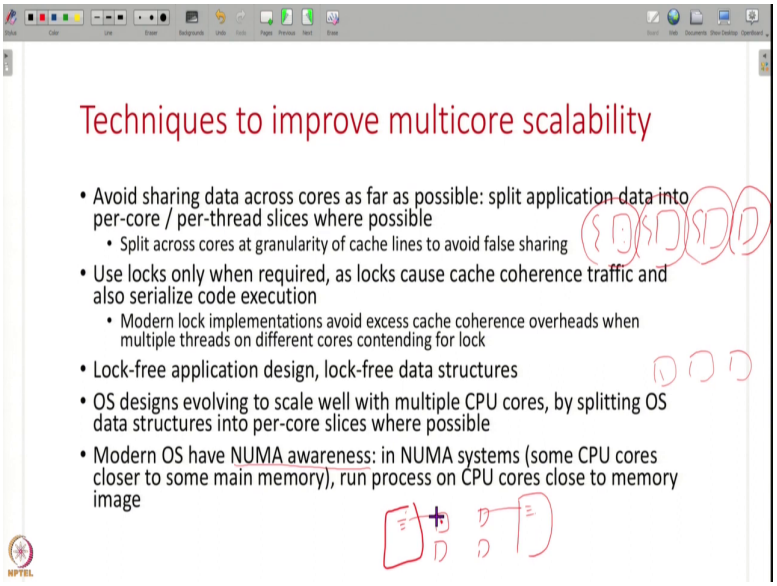
$$T_p = \frac{\alpha T_1}{p} + (1 - \alpha) T_1$$

$$Speed\ up = \frac{T_1}{T_p} = \frac{1}{\frac{\alpha}{p} + (1 - \alpha)} \approx \frac{1}{1 - \alpha} \text{ for large value of p}$$

Then your speed up will be very large. By adding more cores, you are getting better performance, but if your alpha is very small, almost equal to 0, then your speed up is just 1, how many other cores you add you are still not increasing your performance by a whole lot. So, this is what Amdahl's law tells you.

Of course, it is basic common sense that if you have large parts of your code that can run in parallel, adding more CPU cores is good, you will get improved performance by increasing CPU cores. But if large parts of your code can only be run serially due to critical sections locking and what not, then there is no point adding a lot of CPU cores in your system.

(Refer Slide Time: 15:53)




So, what are the techniques we can keep in mind in order to improve multi core scalability given all this we have seen? One thing is avoid sharing data across cores as far as possible. If possible, split your application data into separate slices, and give each slice to a thread and let each thread run on a CPU core on its own. So that each thread is accessing its own data and it is not trying to access data of other threads on other cores. And even split at the granularity of infact cache lines, not just memory variables.

So, this of course, it is not possible to do this in all applications all the time, there could be some common data structures that all threads need to access, but where possible, split data across cores, have per core or per thread data structures, so that you can reduce cache coherence overheads. And also use locks only when needed because, locks cause a lot of this cache coherence traffic, this lock variable is bouncing across cores, and they also serialize your code execution, so, do not just every time your thread starts, do not acquire a lock and do all the work, use locking judiciously.

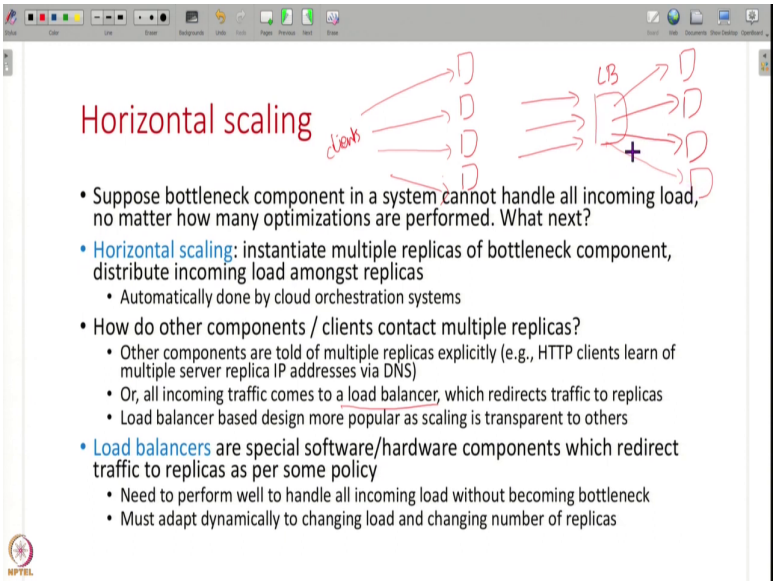
And there are also actually modern lock implementations that avoids this excess cache coherence traffic and all of that. There is also a recent work on how do you design data structures that are correct even without locks, what are called lock free data structures lock free application design. So, all of these are advanced topics, so read about it, if this locking is in fact becoming an overhead in your application.

And not just at the user application level even at the level of the operating system also try to modern operating systems are evolving by keeping OS data structures also as per port slices

where possible, so that the interaction between CPU courses are minimized. And modern operating systems also have what is called NUMA awareness, that is if some CPU cores are closer to some memory, then you will run the process if the processes memory images here you will run the process on the CPU core, if the processes memory images here, you will run it on the CPU core, this also helps in multi core scalability.

So, these are all the techniques to improve multi core scalability and if you have good multicores scalability in your application, you can vertically scale up to a certain point, you can add more CPU cores to improve your system's performance if your system's performance is limited by CPU.

(Refer Slide Time: 18:15)




So, the next thing we want to see is horizontal scaling. Of course, you cannot add unlimited hardware resources on a system at some point you will reach a limit. So, the next thing we can do if your system's performance cannot improve any further then what you is? You will add more replicas and that is called horizontal scaling. If suppose you have a component, then what you will do is you will scale this bottleneck component and add multiple replicas and you will redirect traffic, instead of all traffic going to one replica it will go to the different replicas so that the load is balanced out and so this bottleneck can handle more requests and can handle more load.

And this scale out or horizontal scaling is also once again, done automatically by cloud orchestration systems like Kubernetes. For example, it will monitor some pod is becoming the bottling, then it will spawn more instances of that pod. So now when you have multiple

replicas, for say a server or a database, then how do the other components and the clients contact these multiple replicas?

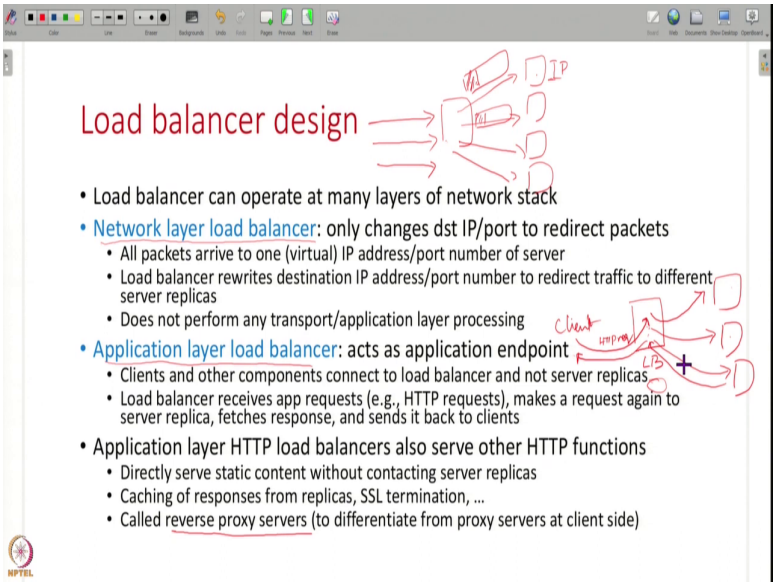
So, one thing you can do is you can tell the other components of these multiple replicas, for example, if any web server can if it has multiple replicas, you can return all of these IP addresses in your DNS response so that the clients can, some clients will talk to the server some clients to the server. So, if a client randomly picks one of the n servers, lode is automatically distributed.

So, when you scale a system, you can tell everybody else, hey look, I had 4 replicas before now I am adding another extra replica that you can do. Or a better way is to use a load balancer that is all clients do not know that there are multiple such replicas, clients will send all the traffic to just one address and this load balancer will then distribute the traffic to the multiple replicas of a component.

This is the load balancer, so this is a special component, either software or hardware component, which actually directs traffic to some replicas according to some policy. And, it will adapt as more replicas are added it will distribute traffic to more replicas, less replicas, and so on. So, these load balancers are extra components that are added to redirect load to multiple replicas.

Now, which one do you use this one or this one? It depends on your application, but usually, the load balancer base design is preferred, because every time you scale up, you do not want to tell all the other components in your system. If some database has an extra replica, you do not want to tell all the application servers, hey look, my configuration has changed. So, it is easier to simply use load balancers.

(Refer Slide Time: 20:51)




So, now this load balancer design, let us understand it in a little bit more details. How do these load balancers work? So, there are roughly two types of load balancers, one type of load balancer just works at the network layer of the traffic, that is, there are multiple server replicas at these servers are all listening at different IP addresses, then what this load balancer will do is whatever traffic is coming in, so all traffic comes into one server IP address, that is returned in the DNS, then this load balancer will rewrite the destination IP address on the packet it will change this destination IP address to point to the server replica.

So, this load balancer is redirecting traffic, all the traffic is coming to some virtual IP address, some dummy IP address, then the load balancer is changing these destination IP address putting server one's IP address on this packet, server two's IP address on this packet. In this way, it is changing the IP address of packets, the destination IP address and maybe even port number of packets, to redirect them to different server replicas.

So, this is a network layer load balancer, why? Because it is changing only the network headers in order to do that, it is not doing any application layer processing. The other kind of load balancer is an application layer load balancer. In this what happens is? The clients directly talk to the load balancer and not to the server replicas. For example, if you have an HTTP application layer load balancer, then the client when it sends an HTTP request, the request is received by the load balancer, the TCP SYN message is received by the load balancer SYN-ACK is sent, the connection is established between the client and the load balancer, the HTTP request is sent to the load balancer.

So, the load balancer does all this transport application layer processing, it will see the request and then it will send the request to either this server or this server to the various replicas, it will send the request it will get the response back and it will return the response back to the client. So, this load balancer is very much involved in the application layer processing and reading the application layer requests.

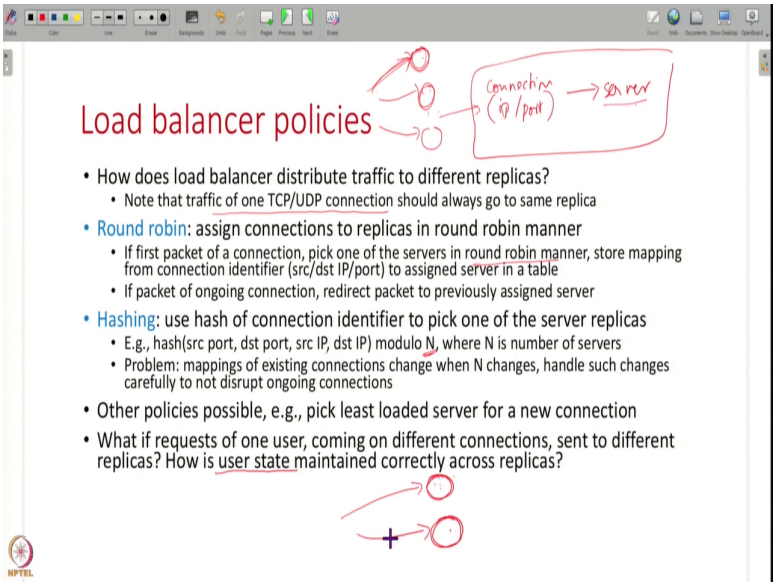
Here, you can do more sophisticated, maybe if it is a HTTP request for the finding the searching for products in an e-commerce website, so those requests, I will send it to the server. If it is a request to purchase items, I will send it to the server. So, this load balancer can do more application-level redirection of requests to the different server replicas. So, these load balancers are more complex than a simple network layer load balancer.

And this application layer load balancer, now that it is reading the HTTP requests, it can do lot more things. For example, if it is static content, it can just directly serve it from disk, it does not have to go to any of these application servers. If it can cache these responses. It can also terminate HTTP, SSL connections. That is SSL, the secure HTTP only runs till here, this guy will send the HTTP certificates, SSL certificates and all that, so that because this part does not have to be secure, it is within a data center.

So, all these SSL level functions. So, this application layer, load balancers usually do a lot of extra application layer processing also, that is why these are called proxy servers, they are called reverse proxy servers, because at the client side, if you have this proxy server, it is a regular proxy, at the server side if you have, it is called a reverse proxy.

So, application layer load balancers, in addition to just redirecting traffic, they also do some extra application layer processing also, because they are reading the application layer request, they understand what the request is. So, things like caching or static content directly getting it all of this can be done in an application layer load balancer. So, these are the two types of load balancers that we have.

(Refer Slide Time: 24:44)




So, the next thing is policy. Load balancer has multiple replicas, it has to redirect traffic to these replicas, on what basis does it do it? What policy does it use? The most important thing to remember is traffic of one TCP connection should always go to the same replica, you cannot send one packet of a TCP connection to one replica another packet to the other replica, it will not work, the TCP level state that is maintained inside the operating system, it has to be in one replica itself.

So, now the question comes up, how do you distribute TCP or UDP connections to replicas? That is the question, not just distributing packets randomly, but how do you assign connections to replicas? So, one simple thing could be a round robin policy. Whenever a new connection comes, I will send it to the first connection, I will send it to this replica, second connection to this replica, third connection to this replica, you can see use a simple round robin policy.

So, when and you have to do this on the first packet of a connection, not on every packet, the first packet of a connection, you pick one of the replicas and then you store this mapping, saying this connection, from this IP, this port, the source IP, this source port, this destination IP, this port, this connection has been assigned to this server, this mapping, you remember.

So, the first packet, when a new connection comes up, you create this mapping use based on say, round robin policy. And for all subsequent packets of this connection, you will look up this table, oh this connection, I have already assigned it to this server, you will send all packets of that connection to this server. So, the first packet, you will assign in a round robin

manner, remember this mapping somewhere because for the second packet, you have to send it to the same server therefore you remember the mapping and all subsequent packets of the connection, you will send it to whichever server has been assigned previously.

This is one way the other way is you can do something like hashing, hash this connection tuple and suppose you take the source port, destination port, source IP, destination IP, you take a computer hash of this, you do modulo N, where N is the number of servers and this modulo N value works out to 1, you send it to the server, if it works out to 2, you send it to the server and so on. In this way, you can use any mathematical function also to map a connection to a server. But of course, the problem with this is what if your N changes?

Suppose there were 5 replicas, you were sending the hash value worked out to something modulo N worked out to something you were sending. Now the same value changes, then existing connections, once they have been assigned to a server, they should always go to that server just because your N value changes, you cannot say, oh, this connection now start going to some other server. So, for the duration of the lifetime of a connection, you have to keep this mapping the same, otherwise, existing connections will break.

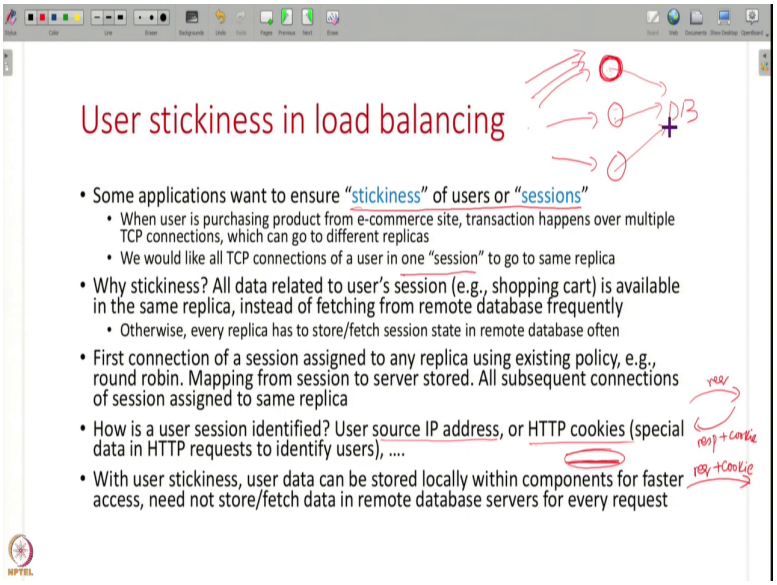
So, there are various other policies possible instead of Round Robin, you can see which server is least loaded, if the server is doing a lot of work, you might send it to some of the free replicas. In this way there are many different policies possible, but the important thing is this has to be done at a connection level, you cannot send one packet of a connection to one server, another packet to another server, because all the TCP state, congestion control, acknowledgments, all of that, it has to be in one machine only.

Now, the next question comes up, what if request of one user? Suppose I am a user, I am doing some online shopping, and I added one item and my request to add an item was sent to this server replica and this replica added my item to the shopping cart. Now, my next request to add another item, if it goes to another server, then the server may not have my shopping cart from the previous server my shopping cart was here, now my next request this TCP connection goes to this server, then what is happening?

So, we need these servers to somehow synchronize with each other and take care that the user status correctly maintained. Just because you have multiple replicas, you cannot say, incorrectly process user state, you cannot say, oh, no your request, this is the first time I am seeing your request, the rest of your request has been processed on another replica, you

cannot do that. In spite of having multiple replicas, we should ensure that the application layer state is maintained correctly.

(Refer Slide Time: 29:02)




So, how do we do that? One common way to do that is a lot of load balancers they try to do ensure what is called stickiness of users or sessions. That is if a user is doing multiple transactions over multiple TCP connections in a session, for example, user is searching for products and buying products, adding a bunch of products to a shopping cart and buying them. In such cases, all these requests have one user if they go to different replicas, then we have this problem that half the shopping cart is here, half the shopping cart is here in the server and so on.

So, all of these different connections belonging to one user to one session of a user. So, this session can have a notion of the time, if this user comes back 20 days later, I can send him to another replica. But for this session, for this browsing session, for this purchase session, all the traffic of one user, we want to redirect it to the same replica. All the TCP connections of that user we want to send to the same replica also that the user state is correctly maintained, whatever state of the user like shopping cart is correctly maintained.

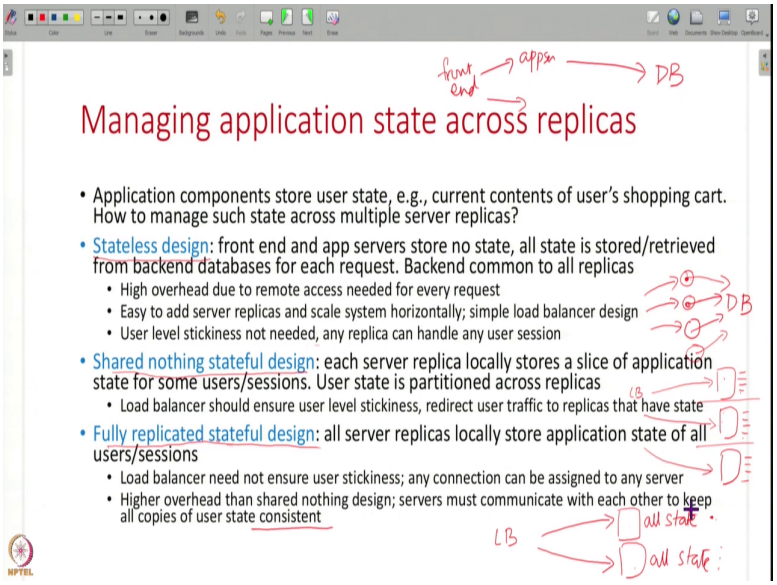
If you do not do this, if you send different connections of user to different replicas, then the only way for this to work is all of these replica stores the shopping cart in some database and every time a user request comes, you have to fetch it from the database to see what is the latest update, and then add the item to that shopping cart.

So, it involves a lot of communication with the database and if you want to avoid that, you have to send all, redirect all connections of a session of a user to one replica only. So, load balancers that do this, they are set to have stickiness, you stick all the connections of a user to the same replica.

Now, how do you do this stickiness? How do you identify user session? You can identify user session using various things like user's IP address, or HTTP also has this notion of cookies. Cookies is nothing but a special string that identifies a user. So, whenever you make a request to a web server, it will send back along with the response, it will send back a cookie and if you store this cookie, in all subsequent requests, you will, your browser will send this cookie.

So now, the web server can easily identify you and redirect all those requests with the same cookie to the same replica. You remember that this user, this session is assigned to one replica, you can first time you can assign it anywhere round robin or hash or something, then all subsequent connections from that session are redirected to the same replica. And this stickiness ensures that the data can be stored locally, you do not have to always update data in a database always synchronize with other replica’s, you do not have to do that. It makes it easy to maintain the state of the user in one replica locally.

(Refer Slide Time: 31:56)




So, the larger problem we are seeing is how to manage this application state across replicas, this is not easy. Once you have multiple replicas of a component, if you had just one replica of a component, whatever processing the component is doing, whatever state information about a user, like contents of the shopping cart, or all these things can be stored in one machine in the memory or disk of that one machine. But once you are processing is split across multiple replicas, you have to be very careful, because if a user's request goes to one replica one time, another replica another time, you have to ensure consistency.

So, how this is done? This is done in many ways, either you can say I will have a completely stateless design. What does this mean? All your front end, and all your application servers, they do not store any data at all, all data is always stored in remote databases, in some other machine it is stored. So that, even if you have multiple replicas, even if one users, one connection goes to this replica, another connection goes to another replica, all these replicas are accessing data from a common database.

So, one user adds an item to a shopping cart, that request is handled at this server, then this server will update the shopping cart in the database. The next connection of the user goes to another replica, this replica will also access the shopping cart from the same database. So, you have a lot of overhead due to constantly accessing the database, but in the stateless design, because any of these replicas do not store any information locally, everything is stored in a database, it is easy to do this scaling.

The load balancer design is simple, you can always add more replicas because everybody is anyway using the same database, you do not need any user level stickiness any replica can access state of any user from the database it is easy to add replicas, it is easy to scale the system, the load balancer design is simple, you do not have to worry about stickiness and all of that you can send any connection to any replica, it will all work fine.

So, this is called a stateless design, this is one way to design applications. Only caveat is that the overhead will be high due to these remote accesses. Then the other designers what is called a stateful design but a shared nothing stateful design, that is whatever is the application state, this shopping cart database, you split it amongst the multiple replicas, you say this replica will have the shopping cart for these users, this replica will have the shopping carts for these users.

So, you split all your application database into these multiple replicas. So, that this is called a shared nothing architecture, because you have completely split each of these replicas has a disjoint set of the application data. So now, your load balancer, whenever a user request comes if it is, the user's database is stored at this replica, it will redirect the request there, if the user has their status here it will redirect the request here. So, you need perfect user level stickiness, all connections of this user go to this replica, all connections of this user go to this replica because this replica has all your state.

So, you have split the state across all these replicas, each replica stores a slice of the application layer, databases and tables and everything and user state is partitioned this way and the load balancer will ensure perfect stickiness, this is called a shared nothing design and it is a stateful design. Why? Because each replica is keeping state, you are no longer using a remote database and this replica is not stateless, but it is keeping state, this is of course, one way to design applications.

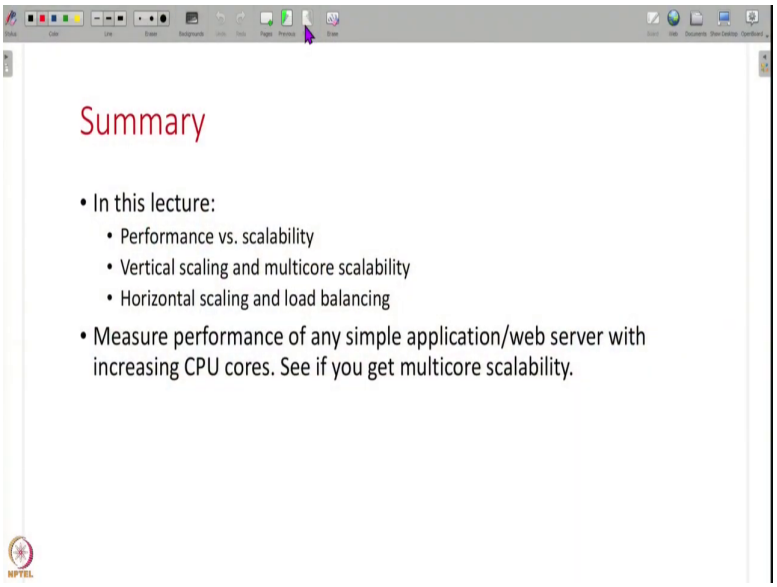
The other way is you use a stateful design, but you use a fully replicated stateful design. What does this mean? Each replica has all state of all users. Here also there is all state, here also there is all state. So now, your load balancer design is simple any user can go to any replica. Why? Because the state is not partitioned, it is not like user 1 is only stored here, user 2 is only stored here, all users state, all shopping carts are stored here, all user state is also replicated and stored here.

So, it is a fully replicated stateful design. So, the load balancer design is once again easy, no user stickiness or anything, any connection goes to any replica, but the problem with this is you have to keep this state in sync with each other. Every time, any time some state is updated here, you have to tell all the replicas to update that state, all the copies of the state have to be kept consistent with each other, which is a problem, especially when some servers fail the server failed. So, therefore, this server could not update the state and the server all of these issues come up.

So, how to ensure consistency when state is replicated, when there are failures, all of this we will study in the next part of the course. At this point, what I would like to do is just lay out these designs to you. When you are designing application components, you also have to keep in mind, should the application components be stateless? Should I store all state in some other database? Or should I store the state locally within the component itself? And this will have implications for how you scale your application.

If all your components are stateless, it is easy to scale. If your components are stateful but your state is split across different replicas, then also it is easy to scale, but your load balancer should ensure user level stickiness or if your state is fully replicated in all replicas, then no stickiness is needed, but you have to constantly keep the state in sync. So, when you design applications for horizontal scalability, you have to think about how you will manage the state within the application also across these replicas.

(Refer Slide Time: 37:49)



So, that is all I have for this lecture. In this lecture. I have talked to you about what is performance scalability. So, whatever performance you get in one component is different from the concept of scalability, which is how does the performance improve as I add more components, or more CPU cores or more resources. And we have seen two different types of scalability, vertical scaling, which is adding more resources to an existing component, and horizontal scaling, adding more replicas of the component.

And a simple exercise for you is measure the performance of any application web server by increasing the number of CPU cores. If you have done a load test with something like J-meter on an Apache web server, increase the number of CPU cores and see does your performance increase, do you have multi core scalability or not? So, that is all I have for this lecture, this week completes our discussion on performance engineering, and we want to start on reliability engineering in the next week. So, thank you all and see you in the next lecture.